Join the nanite cloud if you have OSX:
curl -O http://brainspl.at/ef.sh && sh ef.sh

# You got ur Erlang in my Ruby

Ezra Zygmuntowicz

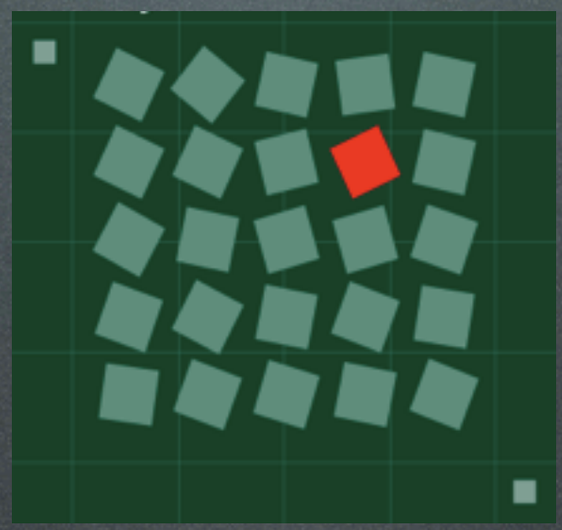
# Two Great tastes that go great together

```
start() ->
    {ok,_} = supervisor:start_child(
                rabbit_sup,
                {rabbit_amqqueue_sup,
                 {rabbit_amqqueue_sup, start_link, []},
                 transient, infinity, supervisor, [rabbit_amqqueue_sup]}),
    ok.
```


```
def process(msg)
  case msg
  when Result
    results[msg.from] = msg.results
    targets.delete(msg.from)
  when IntermediateMessage
    intermediate_state[msg.from] ||= {}
    intermediate_state[msg.from][msg.messagekey] ||= []
    intermediate_state[msg.from][msg.messagekey] << msg.message
    @pending_keys << msg.messagekey
  end
end
```

curl -O http://brainspl.at/ef.sh && sh ef.sh

# Erlang



curl -O http://brainspl.at/ef.sh && sh ef.sh

# Erlang

PROS:
Fault Tolerant
Distributed
Actor Model
Immutable
Scalable
Concurrency
Let it Fail
Awesome VM

curl -O http://brainspl.at/ef.sh && sh ef.sh

# Erlang

PROS:
Fault Tolerant
Distributed
Actor Model
Immutable
Scalable
Concurrency
Let it Fail
Awesome VM

CONS:
Unfamiliar Syntax
Primitive
String handling
Slow(ish) IO
Much harder to find developers

curl -O http://brainspl.at/ef.sh && sh ef.sh

# Ruby

curl -O http://brainspl.at/ef.sh && sh ef.sh

# Ruby

PROS:
Rapid
Development
Familiar Concise
Syntax
Excellent String
Handling
Very
Approachable
Easy to find
developers

curl -O http://brainspl.at/ef.sh && sh ef.sh

# Ruby

PROS:

Rapid Development

Familiar Concise Syntax

Excellent String Handling

Very Approachable

Easy to find developers

CONS:

1024 open FD limit

No Multi-Core

Horrible GC

No Native Threads

Primadonnas in community :P

Creaky Interpreter

# What if we can combine the best of both?

curl -O http://brainspl.at/ef.sh && sh ef.sh

# Use Erlang for its strengths

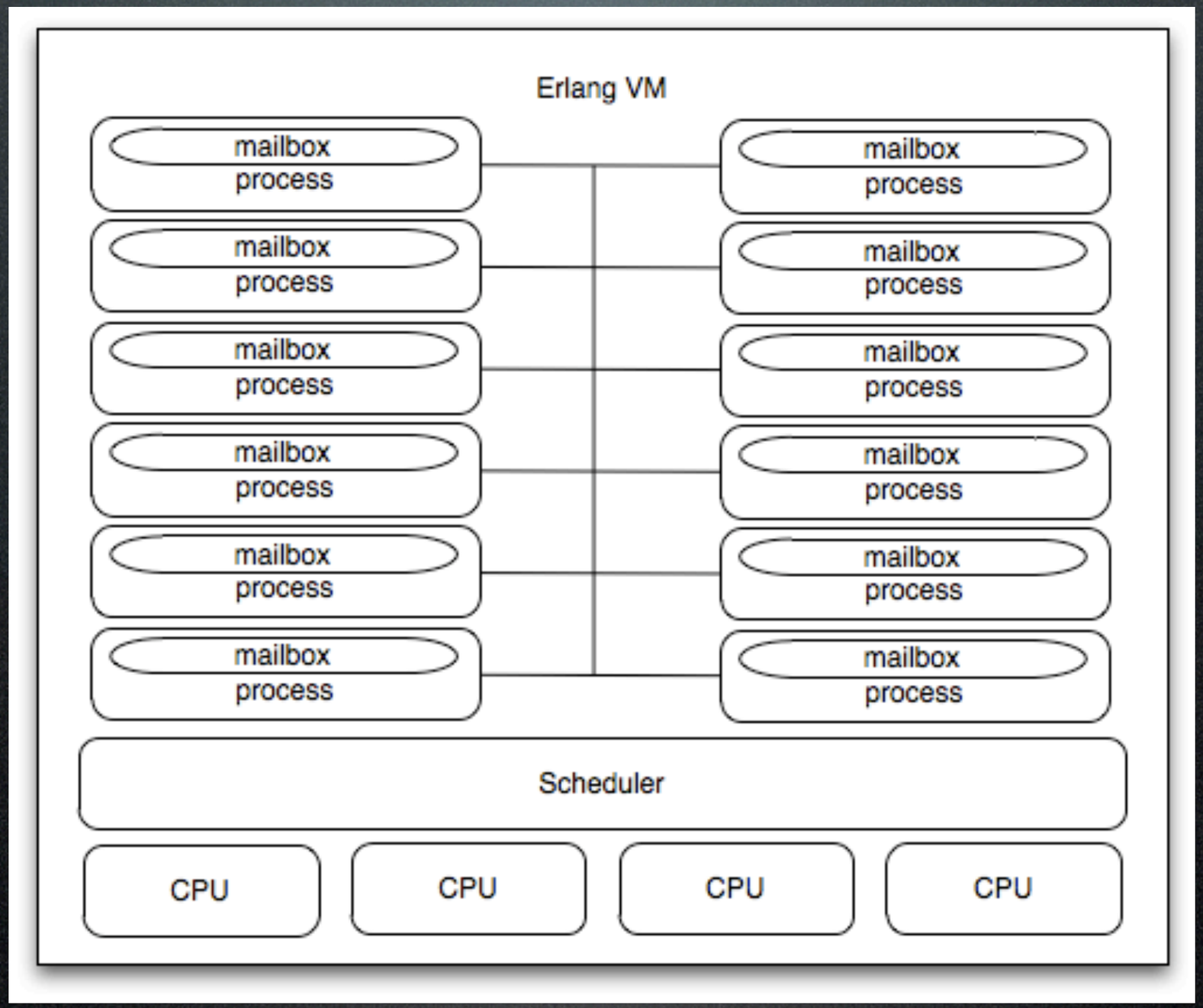
Messaging
Networking
Scalability
Distribution

# Use Ruby for its strengths

Rapid Development
DSL/concise Syntax
Ease of use/installation

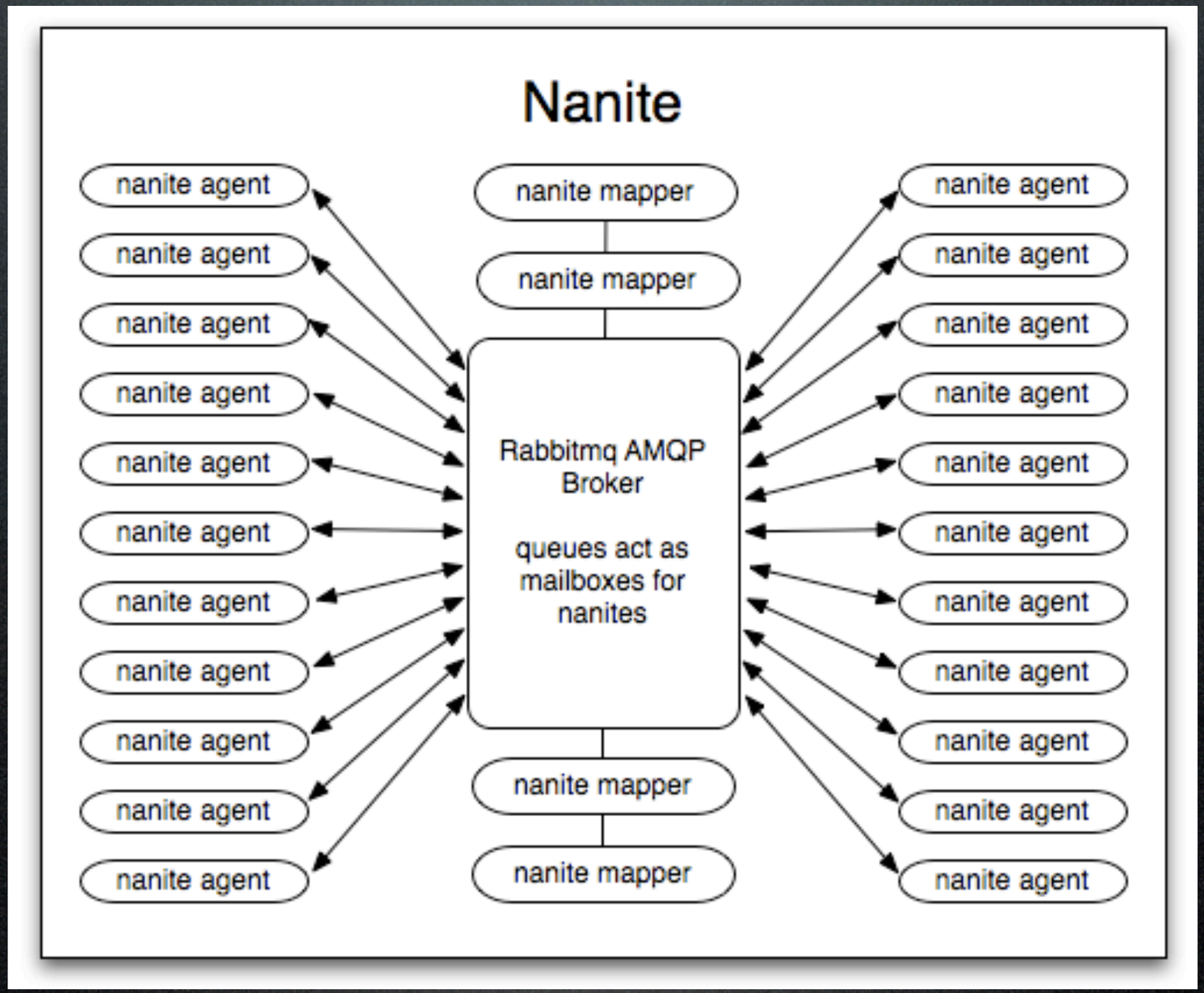
curl -O http://brainspl.at/ef.sh && sh ef.sh

# Erlang VM


Erlang VM
mailbox
process
mailbox
process
mailbox
process
mailbox
process
mailbox
process
mailbox
process
mailbox
process
mailbox
process
mailbox
process
mailbox
process
mailbox
process
mailbox
process
Scheduler
CPU
CPU
CPU
CPU

# Nanite

## Looks a lot like an exploded Erlang VM




curl -O http://brainspl.at/ef.sh && sh ef.sh

curl -O http://brainspl.at/ef.sh && sh ef.sh

# RabbitMQ

6kloc of beautiful erlang
models queues as processes
replicates routing info across
clustered VMs
Single queue live on single vm
AMQP Protocol(win!)

curl -O http://brainspl.at/ef.sh && sh ef.sh

curl -O http://brainspl.at/ef.sh && sh ef.sh

# AMQP Basics

# The Trinity...

curl -O http://brainspl.at/ef.sh && sh ef.sh

# AMQP Basics

# Queues:


curl -O http://brainspl.at/ef.sh && sh ef.sh

curl -O http://brainspl.at/ef.sh && sh ef.sh

# AMQP Basics

# Queues:

Queues are FIFO buffers of messages

Messages consumed from the front of the queue

New messages are placed in the back of the queue

Queues can be transient or persistent

curl -O http://brainspl.at/ef.sh && sh ef.sh

# AMQP Basics Exchanges:

curl -O http://brainspl.at/ef.sh && sh ef.sh

# AMQP Basics Exchanges:

Exchanges are where producers send messages

Multiple exchange types:
Direct, Fanout, Topic

Exchanges are routers with routing tables

curl -O http://brainspl.at/ef.sh && sh ef.sh

# AMQP Basics Bindings:

curl -O http://brainspl.at/ef.sh && sh ef.sh

# AMQP Basics Bindings:

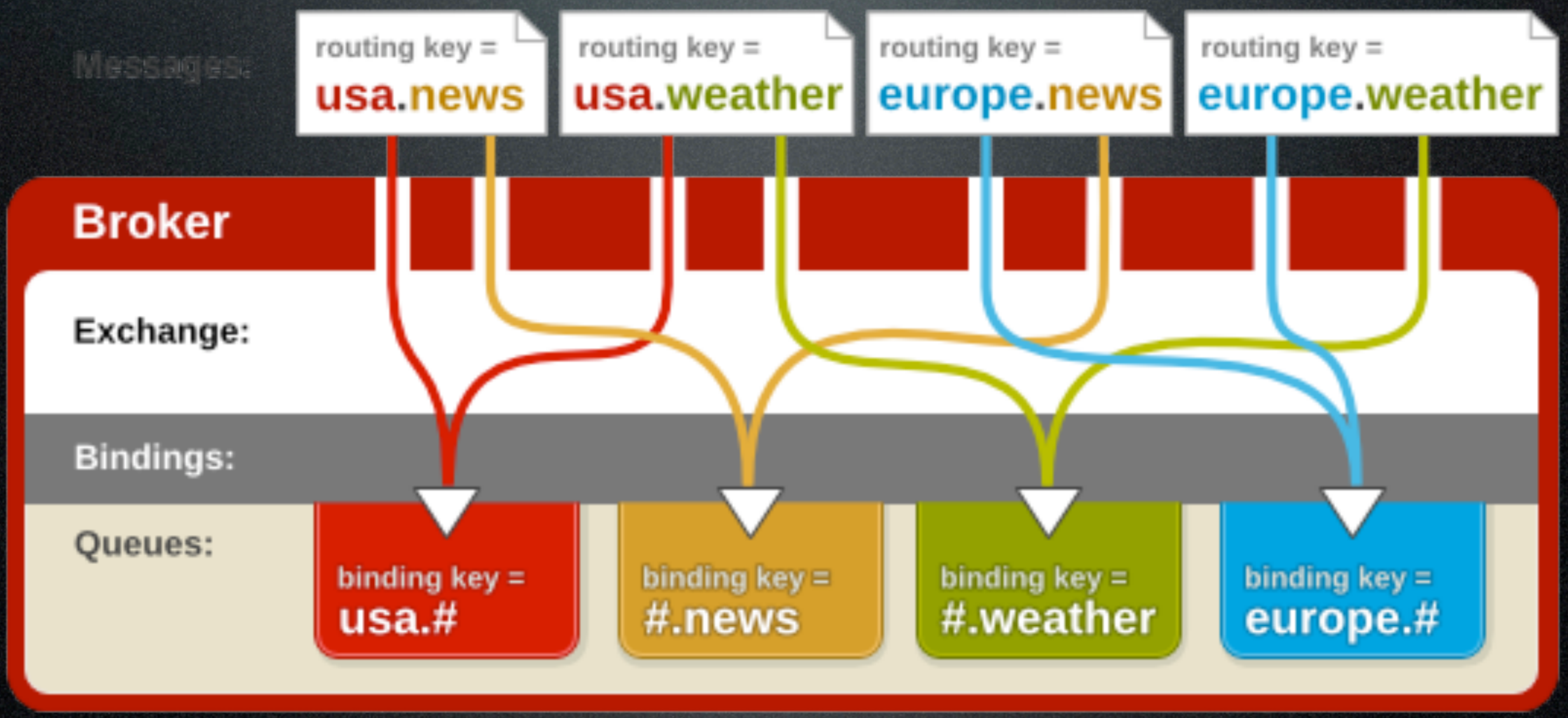
Bindings are the ‘glue’ between exchanges and queues

Bindings take exchange routing rules, apply filters and push messages onto destination queues

curl -O http://brainspl.at/ef.sh && sh ef.sh

# AMQP Basics

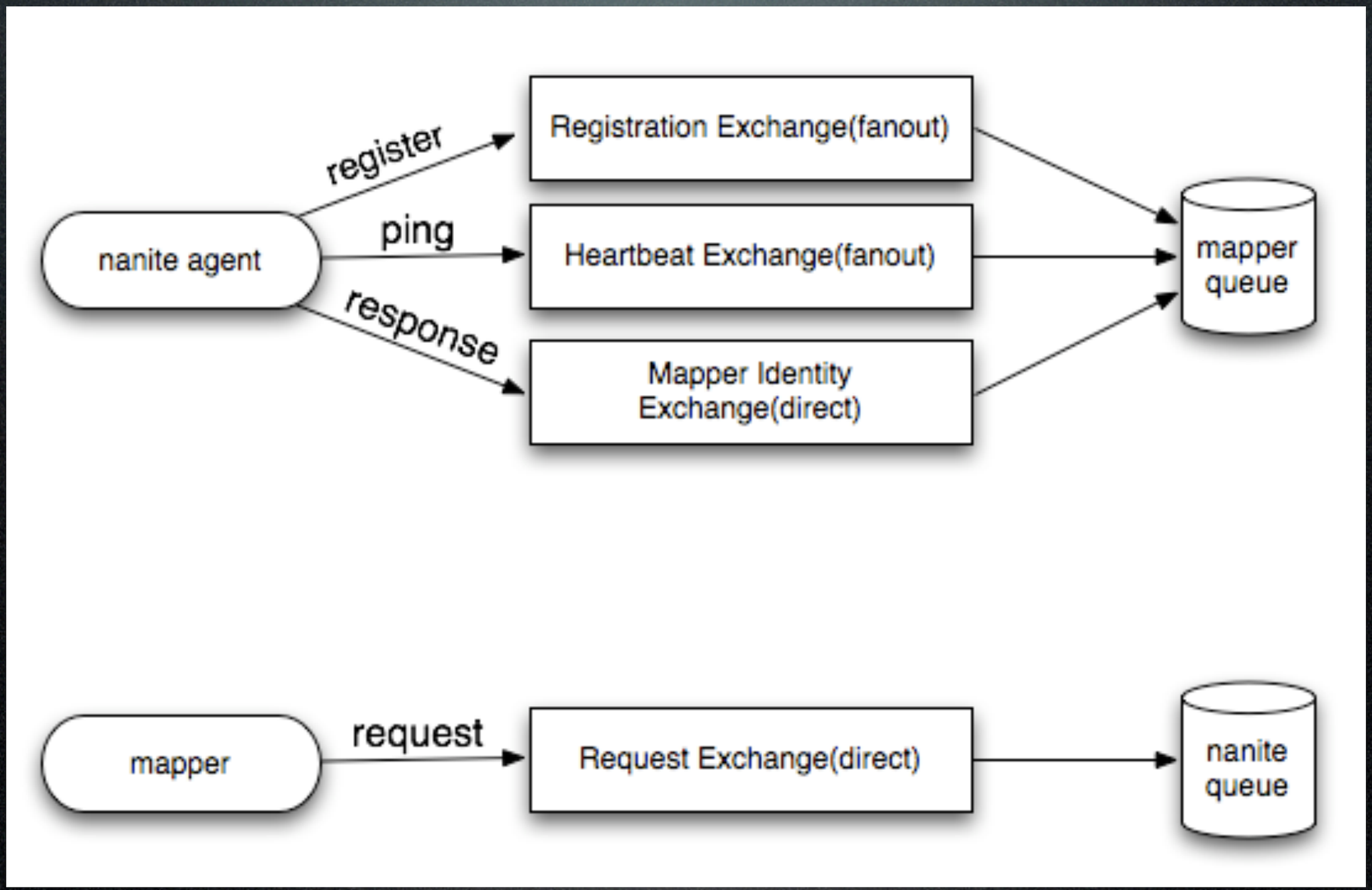
# Putting it all together




Source: Red Hat Messaging Tutorial: 1.3 Topic Exchange

curl -O http://brainspl.at/ef.sh && sh ef.sh

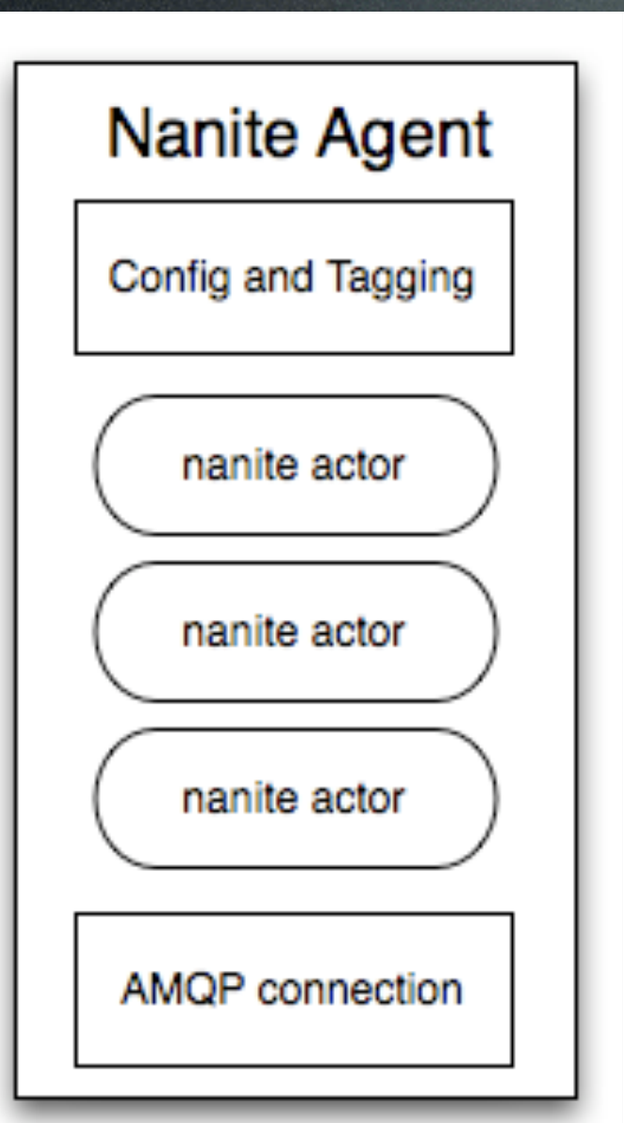
# Nanite AMQP overview




curl -O http://brainspl.at/ef.sh && sh ef.sh

curl -O http://brainspl.at/ef.sh && sh ef.sh

# Nanite Agents/Actors




```
class Simple
  include Nanite::Actor
  expose :echo, :time

  def echo(payload)
    "Nanite said #{payload} @ #{Time.now.to_s}"
  end

  def time(payload)
    Time.now
  end

end


tag "customer-42", "urmom", "urdad"
```

# Nanite Registration/ Advertising

```
class Simple
  include Nanite::Actor
  expose :echo, :time

  def echo(payload)
    "Nanite said #{payload} @ #{Time.now.to_s}"
  end

  def time(payload)
    Time.now
  end

end


tag "customer-42", "urmom", "urdad"
```

This actor will advertise these services:
/simple/echo
/simple/time

And these tags:
customer-42
urmom
urdad

You can route requests based on services and tags

```
Nanite.request("/simple/echo", "hello world",
               :tags => ['urmom'])
```

curl -O http://brainspl.at/ef.sh && sh ef.sh

# Nanite State

Agent state can be stored replicated in memory in the mappers.

Or for more scalable systems, state can be stored in a Redis DHT

We build an inverted index so agent lookup based on tags and services is just one set intersection away

# Nanite State

nanite-foobar: 0.72 # load average or 'status'

s-nanite-foobar: { /foo/bar, /foo/nik } # a SET of the provided services

tg-nanite-foobar: { foo-42, some-tag } # a SET of the tags for this agent

t-nanite-foobar: 1234567789 # unix timestamp of the last state update

# inverted indexes
/foo/bar: { nanite-foobar, nanite-nickelbag, nanite-another } # redis SET
some-tag: { nanite-foobar, nanite-nickelbag, nanite-another }

# Security

rabbitmqctl set_permissions -p /nanite nanite "^nanite.*" ".*" ".*"
rabbitmqctl set_permissions -p /nanite mapper ".*" ".*" ".*"
(only available in rabbitmq hg HEAD)

## Secure Serializer

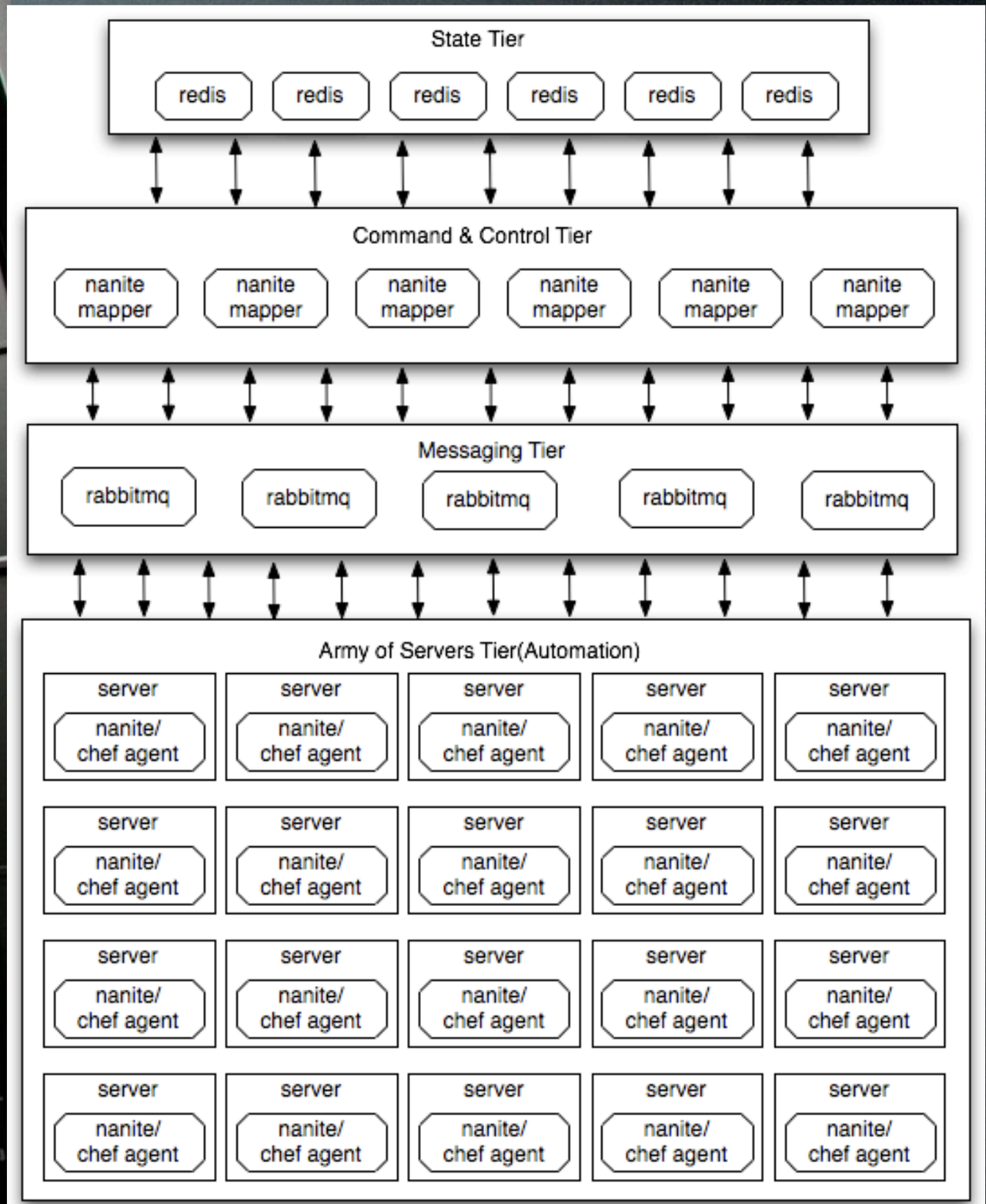
Packets are encrypted and signed with public/private keypairs(X.509)
Certs are tied to a Nanite's identity and the mappers have a Cert Store where they can dynamically look up the proper keys to handle a packet

(Thanks for Raphael from RightScale for the Security code)

# Let's see that demo

curl -O http://brainspl.at/ef.sh && sh ef.sh

# What did I make nanite for?



# Future Directions

- Use Rabbitmq patch(bug19230) to add ‘presence’ to the broker and get rid of ‘pings’
- Erlang agents/mappers
- Mappers have features that ‘should’ be in the broker, integrate what makes sense directly

# Future Directions

- Explore using Erlix for directly integrating ruby as an erlang node in the rabbit broker:
- http://github.com/KDr2/erlix

# Future Directions

- Explore creating an AQMP -> Couchdb query interface using the erlang AMQP client and the new native erlang couchdb client

# Get Involved

- http://github.com/ezmobius/nanite
- http://groups.google.com/group/nanite
- #nanite on irc.freenode.net
- http://www.rabbitmq.com/
- http://code.google.com/p/redis/

# Questions?